MASTER'S THESIS | LUND UNIVERSITY 2015

# Performance analysis and improvement of PostgreSQL

Martin Lindblom, Fredrik Strandin

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-01

# Performance analysis and improvement of PostgreSQL

Martin Lindblom
dt08ml7@student.lth.se

Fredrik Strandin
dt08fs1@student.lth.se

January 21, 2015

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Per Andersson, per.andersson@cs.lth.se

## Abstract

PostgreSQL is a database management system, used in many different applications throughout the industry. As databases often are the bottlenecks in the performance of applications, their performance becomes crucial. Better performance can either be achieved by using more and faster hardware, or by making the software more efficient.

In this master thesis we do a performance analysis of the PostgreSQL database server from the perspective of compiler optimizations, file systems and software prefetching.

We will also show how a data structure used in PostgreSQL can benefit from manually introducing software prefetching, as it is hard for the compiler to predict cache misses and insert prefetching instructions in a profitable way.



**Keywords**: PostgreSQL, postgres, performance analysis, optimization, prefetching

# Acknowledgements

We would like to thank Jonas Skeppstedt for his excellent guidance during our work with this master thesis.

We would also like to thank *Café Finn Ut* for providing proper coffee. This thesis would have never been possible without them.

# Contents

# Chapter 1
# Introduction

In this report we will show how PostgreSQL performs under different file systems, compilers, and compiler optimizations. We will also present a proof of concept of using software prefetching to increase the performance of the PostgreSQL server. This chapter will give an introduction to the report's structure and the subjects we will address in this thesis.

## 1.1 Report structure

We will begin this report with our problem statement that guided us during this Master Thesis, followed by an introduction to PostgreSQL, file systems, optimizing compilers and prefetching.

In the following chapters we will describe our process of evaluation and present our findings, reflections and conclusions. In the appendix we will describe our lab equipment and setup and also present a proof of concept to improve PostgreSQL's performance.

## 1.2 Problem statement

In modern application development, database servers play a crucial part in the overall performance. To allow the databases to perform better, you can either optimize the database server, or modify the environment that the database server is running under.

The environment consists of both hardware, and software in the form of compiler environment, file systems, operating systems, instruction set architecture, et cetera.

To study all of these aspects is not reasonable in a Master Thesis, and therefore we have formulated questions we want to answer. The two main questions we wanted to answer with this thesis are the following:

- Are there possibilities to improve the performance of PostgreSQL without making any changes in how PostgreSQL is designed?

- How much does the following environment aspects affect the performance of PostgreSQL?
  - Compilers
  - Compiler optimizations
  - File systems

We will not go into greater detail about why different specific environments aspects affect the performance in certain ways. The design and implementation of file systems and optimizing compilers are both subjects of their own that take great amounts of effort to study and understand. Our time constraints did not allow us to study these subjects in detail.

## 1.3 PostgreSQL introduction

PostgreSQL is an object-relational database management system and runs on all major operating systems. It implements the majority of the SQL:2011 standard [16] and it is ACID-compliant. The transaction model is based on multi-version concurrency control to avoid locking issues and dirty reads.

The history of PostgreSQL started in the late 1970's when its ancestor, Ingres, began as a project at the University of California, Berkeley. It has since then evolved and became PostgreSQL in 1996 when support for SQL was added. Today, PostgreSQL is an open source project that is maintained by the PostgreSQL Global Development Group, an international group of companies and individuals.

## 1.4 Compiler introduction

In this section we will introduce the compilers used in this study. We will also give an introduction to optimizing compilers, and how these are utilized.

### 1.4.1 Compilers

A compiler is used to turn human readable code into binary code that a computer can interpret. On Linux, which was the operating system used in the lab environment (see appendix B), the most common compilers are *GNU Compiler Collection* [6] (hereinafter called *GCC*) and *Clang* [4] (a C language family front end for *LLVM*).

### 1.4.2 Optimizing compilers

An optimizing compiler has the possibility to introduce changes in the code so that the speed increases for the computations performed by the compiled program. The changes introduced must not affect the result of these computations.

These optimizations are grouped into *optimization levels* by the different compiler vendors, for the convenience of the programmer. The optimization levels can be enabled by using one of the following compiler flags:

`-O1` Enable simple optimizations

`-O2` Enable most optimizations, but without involving a space-speed trade-off

`-O3` Enable all standard compliant optimizations

Both *GCC* and *Clang* are optimizing compilers, and they have the same distinction between the optimization levels. Both compilers also provides the possibility to select individual optimizations from any of these levels by specifying them with compiler flags.

## 1.5 File system introduction

A file system is a way of organizing how data is stored and retrieved. In our experiments we have focused on some popular file systems for locally attached block devices in the form of *hard disk drives* (*HDD*) and *solid-state drives* (*SSD*).

In our experiments we used the following file systems:

- ext2
- ext4
- Btrfs
- XFS
- ZFS On Linux

ext2 and ext4 are file systems of the same family, but from different generations. We were interested in seeing how and/or if the journaling added in ext3 (the generation before ext4) would affect the performance.

Btrfs is a file system developed by Oracle, Fujitsu and Red Hat, focusing on large scale storing. It has support for pooling, snapshots and checksums of data, and uses copy-on-write. It is a relatively new file system, considered stable by multiple Linux distributions since the summer of 2012 [3, 2]. The on-disk format became stable in August 2014 [1].

XFS is a relatively old file system, introduced in 1994 for IRIX. It has for its age support for advanced features like journaling.

ZFS is an advanced file system originally developed by Sun Microsystems, later acquired by Oracle. Due to licensing issues, the original implementation has not been ported to the Linux kernel. Instead a project was started to write a compatible implementation, called *ZFS On Linux* [5]. This implementation is still in its early stages, is not considered fully stable, and lacks optimizations. We chose to include it anyway, to see how it compares to more mature file systems. Throughout this document we will refer to ZFS On Linux as ZFS, if nothing else is stated.

## 1.6 Prefetching

In a modern computer there are multiple kinds of memory at different levels. These memories work with different speeds and are of different sizes. Having the right data in the right place is vital for optimal performance.

The computers used for this thesis (see appendix B) have the following levels of memory:

- CPU internal registers
- L1, L2 and L3 caches
- RAM memory

The number of instruction cycles needed for accessing memory increases about tenfold for every level, starting with one instruction cycle for accessing a CPU register. Trying to load data not stored in registers therefore becomes increasingly expensive depending on where it is first found.

To mitigate this, the system can instruct the memory management to fetch data prior to its usage. This is done either by hardware prefetching, or manual software prefetching.

To increase the perceived speed of a CPU, manufacturers have developed hardware prefetching. Commonly these prefetching algorithms do stride analysis. If data is loaded from RAM with even strides, and cache misses occur, then the CPU prefetches the data with the same stride distance. This obviously only works when data is stored in a consecutive part of memory, with a repeatable pattern between elements, for example in an array [9].

As there are multiple popular ways to store data in memory that do not use even spacing between elements, or are guaranteed to even use the same part of memory, there are software prefetching instructions available for most popular instruction sets/CPU architectures [10].

Adding software prefetching is not always beneficial though. It is hard to time the fetching correctly, and you risk dropping data from the cache that is actually more important to keep cached. The Linux kernel removed their use of software prefetching from their linked list implementation, as it was not beneficial in the majority of use cases [12].

Compilers can also insert software prefetching but they have problems performing analysis of recursive data structures i.e pointer-based data structures such as linked lists, trees, graphs et cetera. The fact that these structures are linked together with pointers is part of the problem. One cannot prefetch a future node before all intermediate nodes and the current node are fetched. This means that it is hard to hide the latency of fetching a future node, and this problem is described as the *pointer-chaser problem* [14].

## 1.7 Benchmarking

When benchmarking a database server there are a lot of functionality, use cases and workflows that could be covered. But as it is not possible, nor effective, to test all possible cases, everyone has to decide which approach is best.

Another aspect of benchmarking databases is standardizations, so the results become comparable between different database servers and engines. *Transaction Processing Performance Council*, hereinafter *TPC*, is a non-profit standardization organization for database benchmarking, founded in 1988. Throughout the years they have published multiple benchmarks of different kinds.

PostgreSQL is distributed together with its own benchmarking program called `pgbench` [19]. It is designed to test a scenario loosely based on the TPC-B benchmark, which is a benchmark designed to measure the throughput in terms of how many transactions per second a system can perform. The TPC-B benchmark is designed by *TPC*.

## 1.8 Related work

Sehgal et. al have evaluated file system performance, and found that *ext2* performs 20% worse than *ext3* and *XFS* when testing database workloads (Online Transaction Processing), due to fact that journaling improves random write performance [15].

Zhou et. al have done a study about the interplay between file systems and SSD performance. In contrast to Sehgal et. al's findings, they found that *ext2* performs well in comparison to *ext3* and that both *ext2* and *ext3* perform well in comparison to other file systems. Their results show that *ext2* performs very well with a database workload, but it is very dependent on block sizes. [21].

The reason behind the difference in *ext2* performance between the reports could be that Zhou et. al tested on SSD disks, which performs well on random writes, and that the benchmarking methods are different.

We were not able to find any related work on the subject of memory cache misses in PostgreSQL.

## 1.9 Contributions

The work on all parts of this master thesis has been equally contributed to by both authors.

# Chapter 2
# Approach

The goal of this master thesis is to analyze PostgreSQL performance and behavior in different environments and see if there is a possibility to improve the performance. We will examine PostgreSQL performance in three different perspectives:

- How optimizing compilers affects code performance
- How different file systems affects PostgreSQL performance
- Can we improve PostgreSQL performance with the help of software prefetching?

In this chapter we will describe how to benchmark PostgreSQL performance, followed by our approach when evaluating PostgreSQL from these previously mentioned aspects.

## 2.1 Benchmarking

We used `pgbench` for benchmarking the database. Each transaction performed contains the following SQL commands:

```
1  BEGIN;
2    UPDATE pgbench_accounts SET abalance = abalance + :delta
3      WHERE aid = :aid;
4    SELECT abalance FROM pgbench_accounts
5      WHERE aid = :aid;
6    UPDATE pgbench_tellers SET tbalance = tbalance + :delta
7      WHERE tid = :tid;
8    UPDATE pgbench_branches SET bbalance = bbalance + :delta
9      WHERE bid = :bid;
10   INSERT INTO pgbench_history (tid, bid, aid, delta, mtime)
11     VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
12 END;
```

The variables in the transactions are decided at random.

The TPC-B benchmark is designed to stress test the core functionality of a database system and one could argue that the benchmark does not apply to the complex database

structures common in the industry of today. We use this benchmark because it is testing the core functionality, as we want to find performance improvements that could be applied to a wide range of the use cases of a database system.

In our lab environment we used two machines. One was used as the database server and the other machine was used to run the `pgbench` program issuing the queries over the network. This setup was used in order to avoid making the client's querying to take CPU time from the database server. In this way we could have a larger number of clients connect to the database server without causing unrelated CPU load for the database server. This approach is recommended by the PostgreSQL Global Development Group [19]. For the hardware and software specifications of the lab equipment see appendix B.

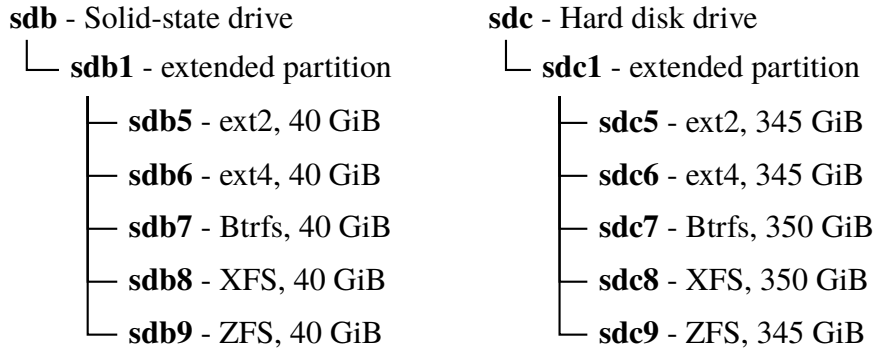
## 2.2 Compiler optimizations

The PostgreSQL Global Development Group recommends GCC when compiling from source, but it is known that PostgreSQL can be built using a variety of compilers [18]. When PostgreSQL is installed from source it is built with the `-O2` flag (see 1.4.2) as standard. This means that the compiler performs most optimizations that do not involve a space-speed trade-off [7].

We want to investigate whether there is a possibility to improve the performance if we compile under another optimization level. It is known that there is no guarantee that compiling under optimization level `-O3` will improve performance. Therefore we will also study how PostgreSQL's performance is affected when individual optimizations from the `-O3` group are enabled. Are there optimizations that have a heavy negative impact on the performance? If this is the case, they could be omitted, allowing other optimizations to improve the performance.

## 2.3 File systems

**Figure 2.1:** Partitions and file systems on Machine 1




The partitions on disks were organized as seen in figure 2.1. The partitions on the SSD held the Write-Ahead Logs (WAL), which were symbolically linked into the corresponding folder on the HDD. Placing the WAL on the SSD avoids making disk I/O a performance bottleneck, as SSD disks are faster than regular HDD's. The PostgreSQL documentation

recommends this approach as writing to the WAL is the main source of PostgreSQL disk I/O [17].

## 2.4 Scaling

We are also interested in seeing how the file systems perform when we scale up the database size. We especially want to study what happens when the database size becomes larger than the RAM memory. Then PostgreSQL must reach disk memory, which will increase the read ratio from disk, and we want to study how the file systems handles these high read rates.

With `pgbench` it is easy to scale up the database size by using the `-s` (scale factor) option. We will test the file systems on four different scale factors:

- 100, which corresponds to a database size of ~1.5 GB
- 1000, which corresponds to a database size of ~15 GB
- 5000, which corresponds to a database size of ~75 GB
- 10000, which corresponds to a database size of ~150 GB

Our lab equipment had a 64 GB RAM memory, so the database will be larger than the RAM memory when we use a scaling factor of ~4200 and larger.

We benchmarked the file systems 8 times om scale factor 100 and 1000, and 4 times on scale factor 5000 and 10000. The reason why we ran fewer times on the larger databases is due to the time needed to construct and set up those large data sets.

## 2.5 Software prefetching

Cache misses are measured with the help of Cachegrind [20]. Cachegrind is a part of the Valgrind tool package and it simulates the interaction between a program and a machine's cache hierarchy. As our target machine has 3 cache-levels (see appendix B), Cachegrind simulates the first-level and the last-level caches [20]. Cache misses on intermediate levels are not simulated.

# Chapter 3
# Evaluation

In this chapter we will present the results of the benchmarks performed in the studies for this thesis.

## 3.1 Performance measurements

We measure the performance of PostgreSQL by running the benchmarking program `pgbench` and see how many *Transactions Per Second* (TPS) PostgreSQL can serve. In order to produce some stable results we let `pgbench` run for 60 minutes, this to allow caches to heat up and to average out noise. This is how the PostgreSQL Global Development Group recommends using `pgbench` [19].

## 3.2 Compiler optimizations

To study how PostgreSQL performs depending on compiler and optimization level, we let the `pgbench` program run eight times per compiler and optimization level. Figure 3.1 shows a box plot summarizing the results for the *GCC* compiler and *Clang* compiler. The tests were carried out using the *ext4* file system.

### 3.2.1 Individual optimizations

To study in more detail how PostgreSQL performs depending on the optimization level we let the `pgbench` program run six times per individual flag from the `O3` optimization level. All of these tests were done using the *GCC* compiler. Figure 3.3 shows a box plot summarizing these results.

### 3.2.2 Cache misses between -O2 and -O3

To further understand the difference in performance between optimization level 2 and 3, we analyzed the amount of instruction cache misses between the levels. Instruction cache misses are a regular problem when using `-O3`. Table 3.1 shows the average ratio of instruction cache misses on the first and last cache levels.

**Table 3.1:** Cache instruction miss ratio for first and last cache levels.

| Opt. level | I1mr/Ir | ILmr/Ir |
|---|---|---|
| `-O2` | 0.0221 | $1.2386e^{-6}$ |
| `-O3` | 0.0233 | $1.3276e^{-6}$ |

The first column in table 3.1 shows the optimization levels. The second and third column shows the ratio of instruction cache read misses on the first and last cache level to the total amount of instructions read, respectively.

## 3.3 File systems

In this test we compiled PostgreSQL with *GCC* on optimization level 2 (`-O2`). Then we ran the `pgbench` program eight times per file system as described in section 3.1. Figure 3.4 shows a box plot summarizing these results.

## 3.4 Scaling

In fig. 3.5 we see the median results of benchmarking the file systems on different scale factors.

## 3.5 Software prefetching

When analyzing the PostgreSQL server with Cachegrind, we found some parts of the source code where cache misses were particularly present. One of the functions where the program had a particularly high amount of cache read misses were `XLogInsert` in `xlog.c`.

By reading through the code it came clear that XLogInsert does a CRC32 checksum calculation on data split up in parts stored in a linked list. Normally the CPU takes care of prefetching by doing stride analysis on cache misses. But because linked lists do not store data with equal distances between elements, it becomes impossible for the stride analysis to predict where the upcoming elements are stored in memory.

Therefore we decided to try and manually introduce software prefetching into PostgreSQL's code in `XLogInsert`. We used the function `__builtin_prefetching` [8] that is provided by GCC and Clang, which on X86 architecture corresponds to the `prefetchtX`-instructions provided by the SSE instruction sets.

In figure 3.6 we can see the impact of this change. There is a $0.54\%$ performance increase on average. A patch-file with the changes is attached in appendix A.1.

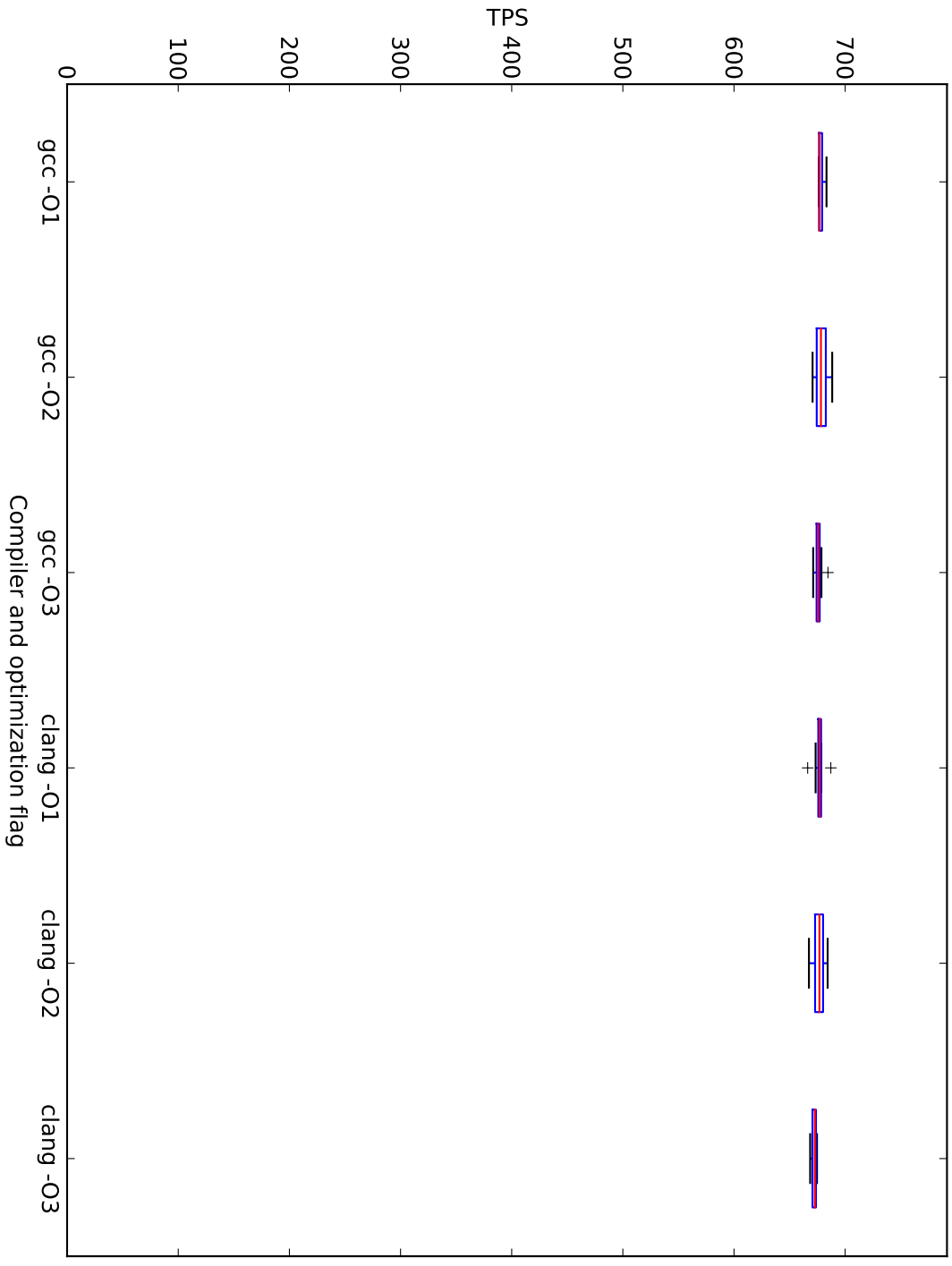



**Figure 3.1:** TPS results when PostgreSQL is compiled with *GCC* and *Clang* on different optimization levels

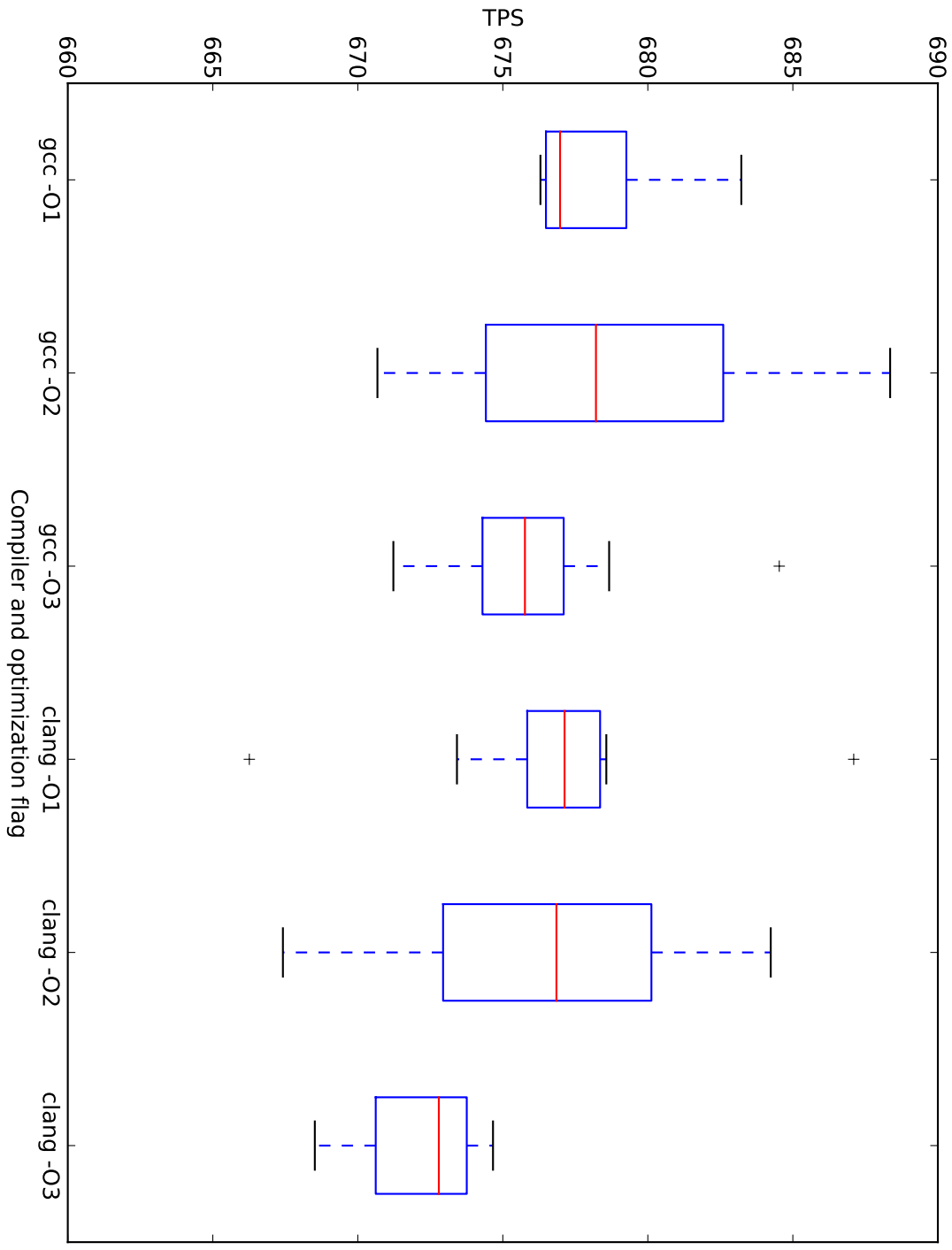



**Figure 3.2:** Same as figure 3.1, but zoomed in.

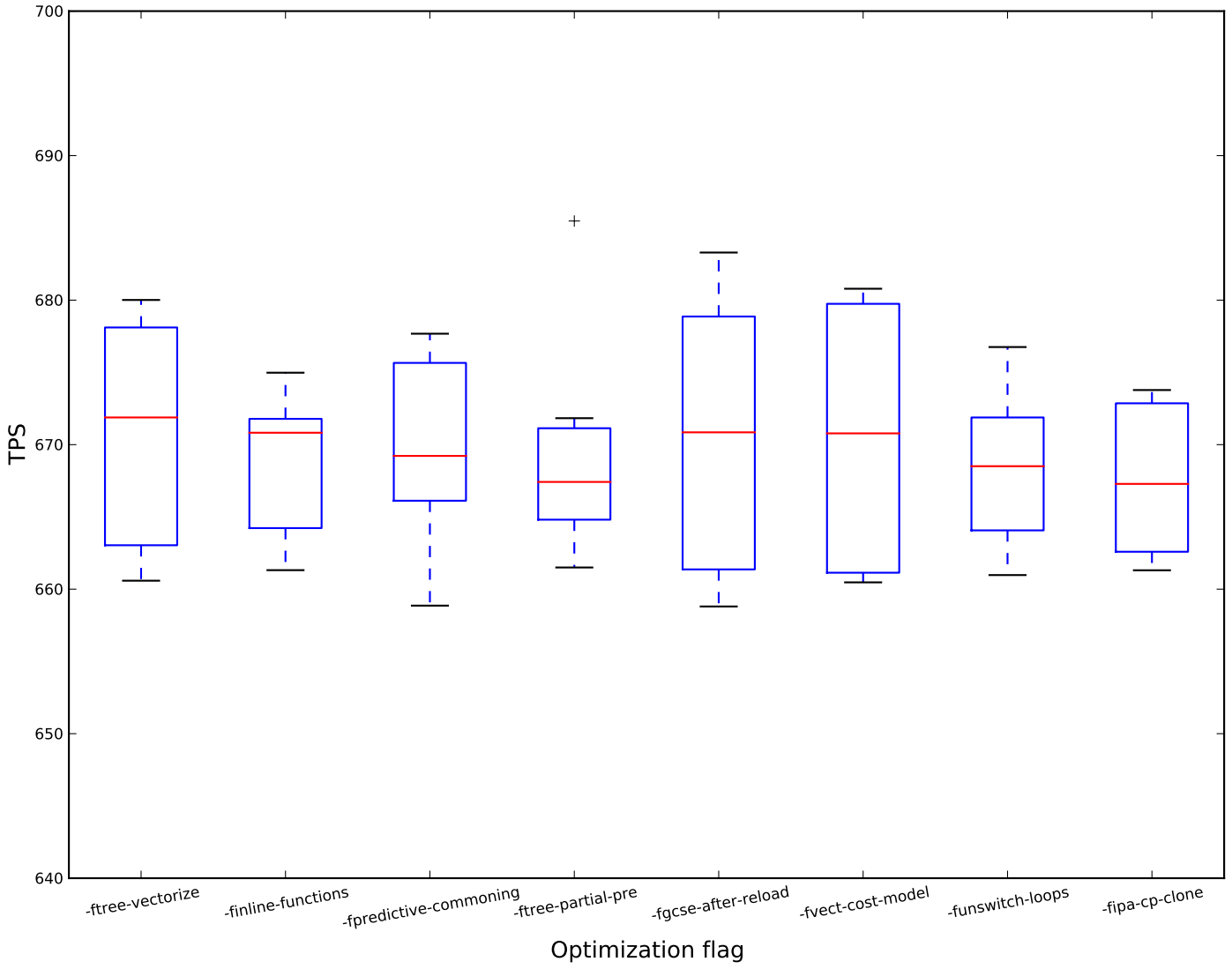



**Figure 3.3:** TPS results when PostgreSQL is compiled under `O2` and individual flags from the `O3` optimization level

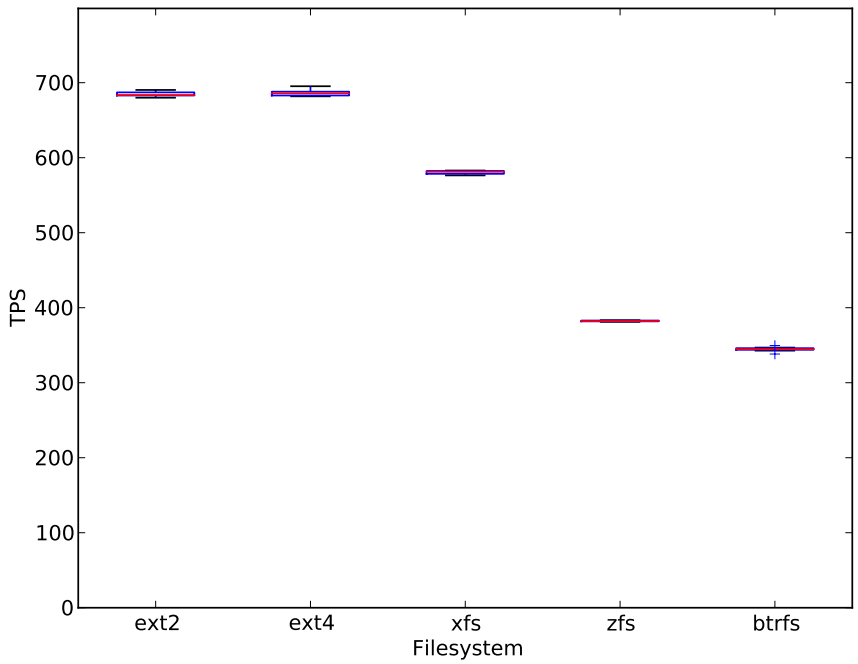



**Figure 3.4:** PostgreSQL TPS on different file systems

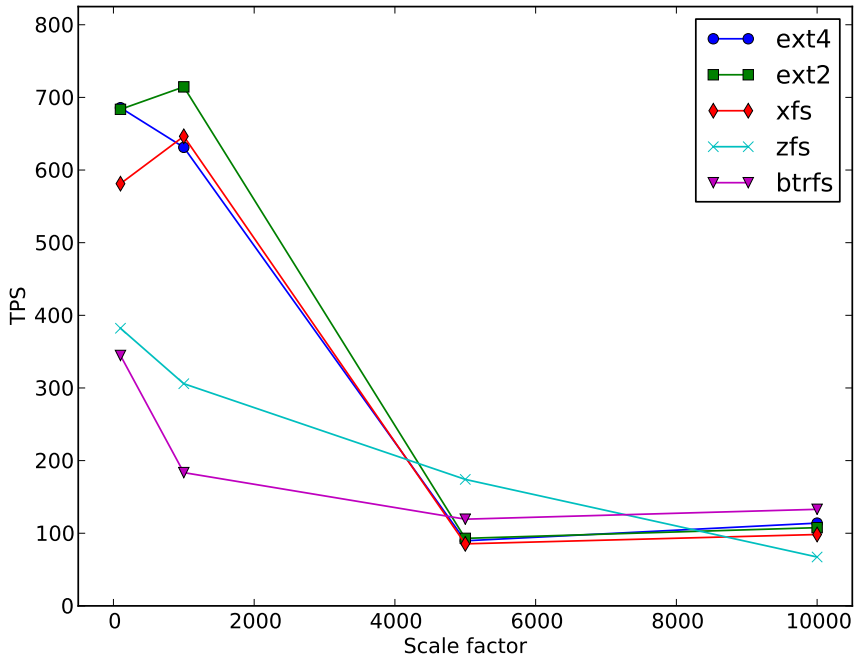



**Figure 3.5:** Median TPS for different file system on different scale factors

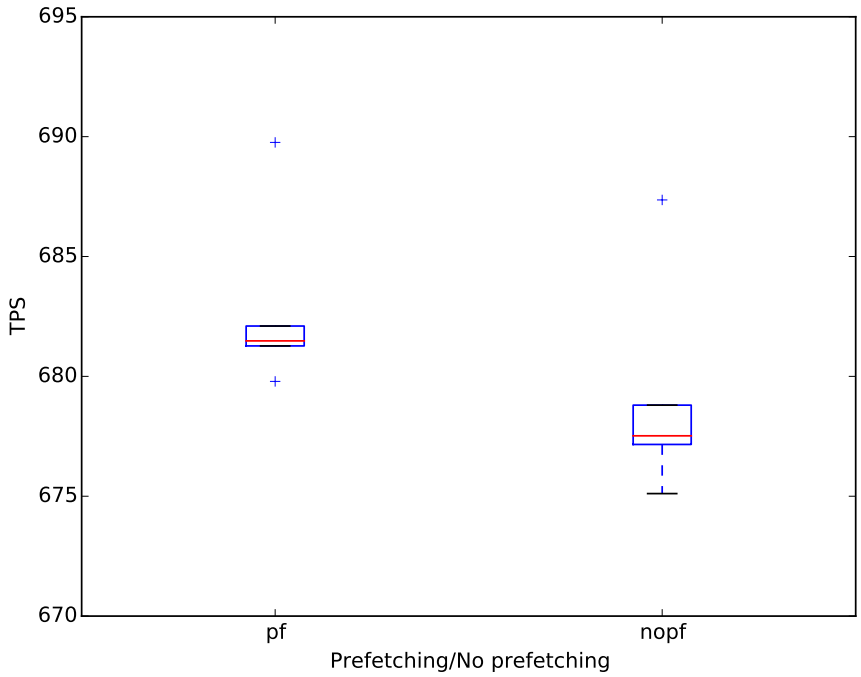



**Figure 3.6:** Prefetching vs. no prefetching

# Chapter 4
# Discussion

In this chapter we will do an in depth analysis of the results presented in the previous chapter.

## 4.1 Compiler optimizations

We notice that both compilers produce similar results. With *GCC* we make a slight performance gain if we compile on `-O2` compared to `-O1`. With *Clang* there is an even slighter difference, but the other way around. All in all the difference between `-O1` and `-O2` is negligible.

There is a performance loss if we compile with `-O3` compared to `-O1` and `-O2`, for both compilers. Especially *Clang* performs badly, with peak performance not even achieving median performance for either `-O1` or `-O2`.

### 4.1.1 Individual optimizations

As we mentioned in the previous section, there is a noticeable performance loss between `-O2` level and the `-O3` level. As we can see in figure 3.3, the median level of the performance is around 670 transactions per second independent of what optimization flag we use, which is lower than if we compile with `-O2` alone.

The optimization with the lowest median TPS is `-ftree-partial-pre`, which is a more aggressive form of partial redundancy elimination (*PRE*). PRE introduces more machine instructions and could be introducing instruction cache misses to the program, decreasing the performance.

Another optimization with bad performance is `-finline-functions`, which is an optimization that declares that the compiler should consider all functions for inlining, and not just the ones declared inlined by the programmer.

Inlining functions means that the code size increases and code sections that used to fit in the cache could now have increased in size, thus causing cache misses and slowdowns. Secondly, as function inlining causes code duplication, this increases the instruction cache usage as duplicate copies may be stored in the cache, causing cache misses. Compilers use heuristics to select which functions to inline, but this method does not work in all cases.

In conclusion, all of the optimizations on level `-O3` decreases the performance of PostgreSQL and it is therefore not possible to omit some of the flags to gain a performance increase. The flags are either decreasing the performance e.g. `-finline-functions` and `-ftree-partial-pre`, or give very unreliable performance results e.g. `-fgcse-after-reload` and `-fvect-cost-model`. The performance loss introduced by `-finline-functions` and `-ftree-partial-pre` are most likely caused by instruction cache misses, as indicated by the data in table 3.1. None of these flags reaches the median TPS we reach if we compile with `-O2`.

## 4.2 File systems

Our study show that selecting the right file system will have an impact on how well PostgreSQL will perform. We see that *ZFS* and *btrfs* perform bad, and the *ext*-file systems perform well.

*ZFS's* bad performance is not surprising. *ZFS* on Linux is a project not yet fully stable, and the original *ZFS* project developed by Sun, was set out to create a file system with focus on data integrity and recoverability, with reduced performance as a trade-off [11].

The other file system that performs poorly is *btrfs*, and one explanation could be that the file system is very new and still lacks some optimizations that the other file systems have.

Our study also shows that *XFS* is outperformed by the *ext*-systems. This is quite surprising as *XFS* is about the same age as *ext2*, and when it came out it was a high-performance file system with features unusual for the early 1990's.

We can see that both *ext2* and *ext4* outperform the other file systems. This could be due to many reasons, for example, *XFS* is older than *ext4* and it has a different kind of file structure. Or, it could be that the *ext* file systems are based on a simpler design and are therefore faster. In order to understand why *XFS*, *ZFS* and *btrfs* are slower than the *ext* systems we have to look into the file system internals, which is out of scope for this project.

In fig. 4.1 we can see that *ext4* performs slightly better than *ext2*. As *ext4* implements journaling and journal checksumming, *ext4* has several advantages over *ext2*, faster recovery, better reliability and as our study shows, better performance. Together with the results for *XFS* (which also implements journaling) we can see that the performance does not depend on whether it supports journaling or not as much as we originally thought.

The great impact the choice of file system has on performance is attributed to the *Durability*-aspect in having an ACID-compliant database server. The database server calls `fsync()` after every change in the database, and before sending back an acknowledgement to the client, to assert durability of the query. `fsync()` is a system call to assert that changes to files are not just written to the RAM-cache, but also written to the actual block device. Normally the operating system does this asynchronically at regular intervals.

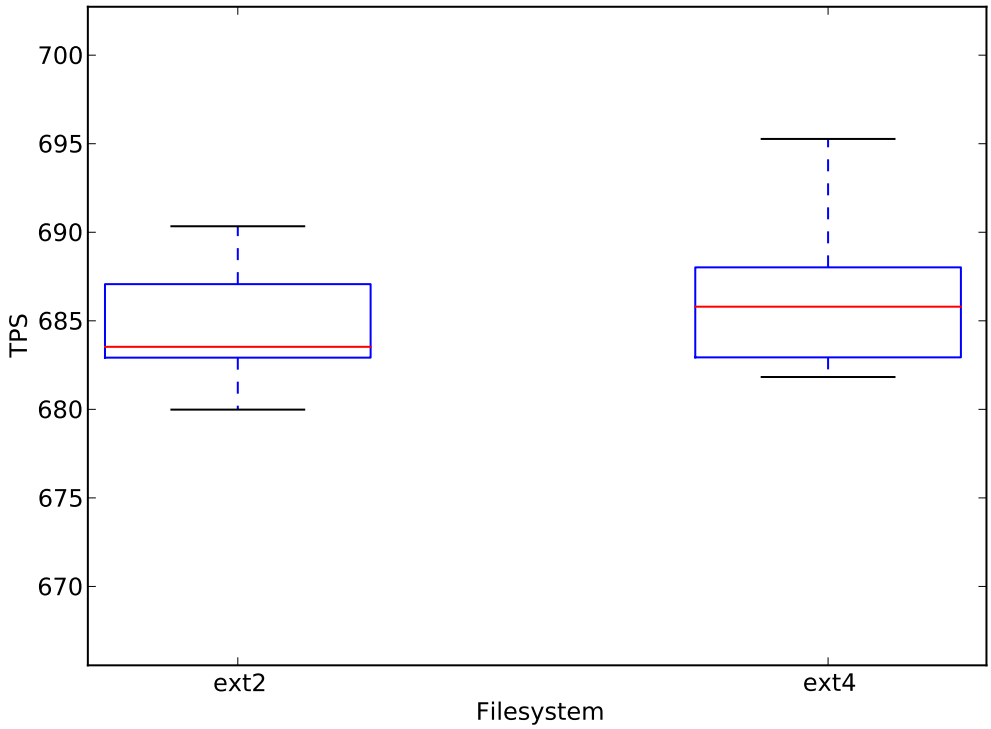



**Figure 4.1:** PostgreSQL TPS performance on ext2 and ext4

## 4.3 Scaling

Our study show that *ext2* and *ext4* perform better when the database is small, and that *btrfs* performs worst. When we use scaling factor 1000 we can observe a large performance increase for both *ext4* and *xfs*. This could be due to that the database size has become large enough to no longer suffer from update contention, that is that transactions are blocking each other when they are updating the database. This is however not the entire explanation as the database should be large enough to eliminate update contention, even on scale factor 100. If this contention existed we would have noticed a similar behavior on all file systems and not only on the *ext4* and *xfs* file systems. Further investigation of this bump in performance on these file systems is out of scope for this thesis.

We can also observe a large performance drop when we increase the scale factor to 5000. On this scale factor the database is too large to fit in RAM memory, so `postgres` must reach disk memory to access database content, which takes up to several milliseconds. We can also notice that *ZFS* performs significantly better than the other file systems on this scale factor.

When we use a scale factor of 10000 we see a slight increase in performance for all file systems except *ZFS*. As we mentioned earlier, *ZFS* is not yet optimized and this could be the explanation behind this low performance on large databases. The explanation behind the slight increase in performance on the other file systems is out of scope for this thesis.

We can also notice that *btrfs* now outperforms all other systems, and it could be due to the fact that *btrfs* is a more modern file system, designed to handle larger amounts of data.

## 4.4 Software prefetching

As seen in the results, software prefetching increases the performance when added in the right place. There are probably more data structures in PostgreSQL that would benefit from this kind of change. But finding places where this kind of optimizations are beneficial is time consuming, and needs great attention to details about the data structures and timing.

Before considering introducing this change upstream, more testing is needed. Prefetching is highly dependent on the hardware, especially the size of the CPU cache. If for example the software prefetching makes other data getting dropped from the cache in an unfavorable way, the change could instead be of harm for the overall performance.

Before we were able to find a beneficial solution, we had to try a couple of different solutions. One of these worked like the one previously used in the Linux kernel [12, 13] for linked lists. In Linux there is an implementation of linked lists, with helper macros for looping over the list. Before Linux 3.0 there was a prefetch-instruction in this macro so that the next item was supposed to be in the cache. But the overall performance was actually degraded. It is still possible to manually prefetch in certain locations if it is predicted to benefit the performance. The default behavior of prefetching was removed in 2011 [13] with the patch in appendix A.2.

The solution previously used by Linux was neither beneficial for PostgreSQL. But as we moved the prefetch instructions to an earlier stage, when the linked list is constructed, the computer got enough time to fetch the data and put it in the cache. During this stage the program had access to all pointers, and could therefore avoid the pointer-chaser problem usual for recursive data structures described in section 1.6.

# Chapter 5
# Conclusions

In the *Problem statement*, section 1.2, we stated the following questions:

- Are there possibilities to improve the performance of PostgreSQL without making any changes in how PostgreSQL is designed?
- How much does the following environment aspects affect the performance of PostgreSQL?
  - Compilers
  - Compiler optimizations
  - File systems

The answer to the first question could be seen as an obvious yes. Most large code bases are open for optimizations, at least for some use cases. This is also the case for PostgreSQL, which we can prove with the proposed introduction of software prefetching in appendix A.1.

To answer the second question, all of the environment aspects mentioned do unsurprisingly affect the performance of PostgreSQL. What we found interesting is how much the choice of file system affects the performance. Choosing the right file system over the wrong can almost double the number of transactions the database server can handle per second.

The compiler and compiler optimizations also affect the performance, but much less than the choice of file system. The single worst choice in regard to these aspects is to use `-O3` optimization.

The best combination of choices in regard to the environment aspects we chose to study are the following:

**Compiler:** *GCC*

**Compiler optimizations:** `-O2`

**File system:** *ext4*

# Bibliography


[1] *BTRFS Wiki Main Page – Stability Status*. https://btrfs.wiki.kernel.org/index.php/Main_Page#Stability_status, (accessed: December 3, 2014).

[2] *Fedora Project Wiki – Btrfs*. http://fedoraproject.org/wiki/Btrfs, (accessed: December 3, 2014).

[3] *Release Notes for SUSE Linux Enterprise Server 11 Service Pack 2 – 3.1.1. Support for the btrfs File System*. http://www.novell.com/linux/releasenotes/x86_64/SUSE-SLES/11-SP2/#fate-306585, (accessed: December 3, 2014).

[4] *clang: a C language family frontend for LLVM*. http://clang.llvm.org/, (accessed: December 6, 2014).

[5] *ZFS on Linux*. http://zfsonlinux.org/, (accessed: December 6, 2014).

[6] Free Software Foundation, Inc. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/, (accessed: December 6, 2014).

[7] Free Software Foundation, Inc. *GCC 4.8.3 Manual Chapter 3.10. Options That Control Optimization*. https://gcc.gnu.org/onlinedocs/gcc-4.8.3/gcc/Optimize-Options.html, (accessed: November 18, 2014).

[8] Free Software Foundation, Inc. *GCC 4.8.3 Manual Chapter 6.55. Other Built-in Functions Provided by GCC*. https://gcc.gnu.org/onlinedocs/gcc-4.8.3/gcc/Other-Builtins.html, (accessed: November 20, 2014).

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual, ch. 3.7.2*. Number 248966-030. September 2014.

[10] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual, ch. 7.4*. Number 248966-030. September 2014.

[11] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, Mark Shellenbaum. The zettabyte file system. Technical report, 2003.

[12] Jonathan Corbet. *The problem with prefetch*. `http://lwn.net/Articles/444336/`, (accessed: December 12, 2014).

[13] Linus Torvalds. *list: remove prefetching from regular list iterators*. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e66eed651fd18a961f11cda62f3b5286c8cc4f9f`, (accessed: December 15, 2014).

[14] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[15] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *FAST'10 Proceedings of the 8th USENIX conference on File and storage technologies*, 2010.

[16] The PostgreSQL Global Development Group. *PostgreSQL 9.3.5 Documentation - Appendix D. SQL Conformance*. `http://www.postgresql.org/docs/9.3/static/features.html`.

[17] The PostgreSQL Global Development Group. *PostgreSQL 9.3.5 Documentation - Chapter 29.5. WAL Internals*. `http://www.postgresql.org/docs/9.3/static/wal.html`, (accessed: November 17, 2014).

[18] The PostgreSQL Global Development Group. *PostgreSQL 9.3.5 Documentation - Chapter 15.2. Requirements*. `http://www.postgresql.org/docs/9.3/static/install-requirements.html`, (accessed: November 18, 2014).

[19] The PostgreSQL Global Development Group. *PostgreSQL 9.3.5 Documentation Chapter G.1. pgbench*. `http://www.postgresql.org/docs/9.3/static/pgbench.html`, (accessed: November 19, 2014).

[20] Valgrind Developers. *Valgrind User Manual Chapter 5. Cachegrind: a cache and branch-prediction profiler*. `http://valgrind.org/docs/manual/cg-manual.html`, (accessed: November 20, 2014).

[21] Ke Zhou, Ping Huang, Chunhua Li, and Hua Wang. An empirical study on the interplay between filesystems and ssd. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, 2012.

# Appendices

# Appendix A
# Code listings

## A.1 Patch for xlog.c

```
diff --git a/src/backend/access/transam/xlog.c b/src/backend/access/transam/xlog.c
index 164b22f..25e410b 100644
--- a/src/backend/access/transam/xlog.c
+++ b/src/backend/access/transam/xlog.c
@@ -838,6 +838,10 @@ begin:;
                                    elog(PANIC, "can backup at most %d blocks per xlog record",
                                             XLR_MAX_BKP_BLOCKS);
                 }
+
+                /* Prefetch data for usage in COMP_CRC32 later on. */
+                __builtin_prefetch(rdt->data, 0, 1);
+
                 /* Break out of loop when rdt points to last chain item */
                 if (rdt->next == NULL)
                         break;
@@ -889,6 +893,9 @@ begin:;
                 rdt->len = sizeof(BkpBlock);
                 write_len += sizeof(BkpBlock);

+                /* Prefetch data for usage in COMP_CRC32 later on. */
+                __builtin_prefetch(rdt->data, 0, 1);
+
                 rdt->next = &(dtbuf_rdt2[i]);
                 rdt = rdt->next;

@@ -906,6 +913,9 @@ begin:;
                         rdt->len = bkpb->hole_offset;
                         write_len += bkpb->hole_offset;

+                        /* Prefetch data for usage in COMP_CRC32 later on. */
+                        __builtin_prefetch(rdt->data, 0, 1);
+
                         rdt->next = &(dtbuf_rdt3[i]);
                         rdt = rdt->next;

@@ -914,6 +924,9 @@ begin:;
                         write_len += rdt->len;
                         rdt->next = NULL;
                 }
```

```
40  +
41  +                   /* Prefetch data for usage in COMP_CRC32 later on. */
42  +                   __builtin_prefetch(rdt->data, 0, 1);
43              }
44
45              /*
```

**Listing 1:** Patch that adds prefetching into the `XLogInsert`-function in `xlog.c`. This patch is based on the tag `REL9_3_5` in the official PostgreSQL git repository.

## A.2 Linux: remove of prefetch

```
1   From e66eed651fd18a961f11cda62f3b5286c8cc4f9f Mon Sep 17 00:00:00 2001
2   From: Linus Torvalds <torvalds@linux-foundation.org>
3   Date: Thu, 19 May 2011 14:15:29 -0700
4   Subject: [PATCH] list: remove prefetching from regular list iterators
5
6   This is removes the use of software prefetching from the regular list
7   iterators.  We don't want it.  If you do want to prefetch in some
8   iterator of yours, go right ahead.  Just don't expect the iterator to do
9   it, since normally the downsides are bigger than the upsides.
10
11  It also replaces <linux/prefetch.h> with <linux/const.h>, because the
12  use of LIST_POISON ends up needing it.  <linux/poison.h> is sadly not
13  self-contained, and including prefetch.h just happened to hide that.
14
15  Suggested by David Miller (networking has a lot of regular lists that
16  are often empty or a single entry, and prefetching is not going to do
17  anything but add useless instructions).
18
19  Acked-by: Ingo Molnar <mingo@elte.hu>
20  Acked-by: David S. Miller <davem@davemloft.net>
21  Cc: linux-arch@vger.kernel.org
22  Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
23  ---
24   include/linux/list.h    | 26 ++++++++++-------------
25   include/linux/rculist.h |  6 +++---
26   2 files changed, 14 insertions(+), 18 deletions(-)
27
28  diff --git a/include/linux/list.h b/include/linux/list.h
29  index 9ac1114..cc6d2aa 100644
30  --- a/include/linux/list.h
31  +++ b/include/linux/list.h
32  @@ -4,7 +4,7 @@
33   #include <linux/types.h>
34   #include <linux/stddef.h>
35   #include <linux/poison.h>
36  -#include <linux/prefetch.h>
37  +#include <linux/const.h>
38
39   /*
40    * Simple doubly linked list implementation.
41  @@ -367,18 +367,15 @@ static inline void list_splice_tail_init(struct list_head *list,
42    * @head:        the head for your list.
43    */
44   #define list_for_each(pos, head) \
45  -        for (pos = (head)->next; prefetch(pos->next), pos != (head); \
46  -                pos = pos->next)
47  +        for (pos = (head)->next; pos != (head); pos = pos->next)
48
49   /**
50    * __list_for_each        -        iterate over a list
51    * @pos:        the &struct list_head to use as a loop cursor.
52    * @head:        the head for your list.
```

```
53    *
54  - * This variant differs from list_for_each() in that it's the
55  - * simplest possible list iteration code, no prefetching is done.
56  - * Use this for code that knows the list to be very short (empty
57  - * or 1 entry) most of the time.
58  + * This variant doesn't differ from list_for_each() any more.
59  + * We don't do prefetching in either case.
60    */
61   #define __list_for_each(pos, head) \
62           for (pos = (head)->next; pos != (head); pos = pos->next)
63  @@ -389,8 +386,7 @@ static inline void list_splice_tail_init(struct list_head *list,
64    * @head:        the head for your list.
65    */
66   #define list_for_each_prev(pos, head) \
67  -        for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
68  -                pos = pos->prev)
69  +        for (pos = (head)->prev; pos != (head); pos = pos->prev)
70
71   /**
72    * list_for_each_safe - iterate over a list safe against removal of list entry
73  @@ -410,7 +406,7 @@ static inline void list_splice_tail_init(struct list_head *list,
74    */
75   #define list_for_each_prev_safe(pos, n, head) \
76           for (pos = (head)->prev, n = pos->prev; \
77  -             prefetch(pos->prev), pos != (head); \
78  +             pos != (head); \
79                pos = n, n = pos->prev)
80
81   /**
82  @@ -421,7 +417,7 @@ static inline void list_splice_tail_init(struct list_head *list,
83    */
84   #define list_for_each_entry(pos, head, member)                                \
85           for (pos = list_entry((head)->next, typeof(*pos), member);        \
86  -             prefetch(pos->member.next), &pos->member != (head);          \
87  +             &pos->member != (head);          \
88                pos = list_entry(pos->member.next, typeof(*pos), member))
89
90   /**
91  @@ -432,7 +428,7 @@ static inline void list_splice_tail_init(struct list_head *list,
92    */
93   #define list_for_each_entry_reverse(pos, head, member)                        \
94           for (pos = list_entry((head)->prev, typeof(*pos), member);        \
95  -             prefetch(pos->member.prev), &pos->member != (head);          \
96  +             &pos->member != (head);          \
97                pos = list_entry(pos->member.prev, typeof(*pos), member))
98
99   /**
100 @@ -457,7 +453,7 @@ static inline void list_splice_tail_init(struct list_head *list,
101   */
102  #define list_for_each_entry_continue(pos, head, member)                 \
103          for (pos = list_entry(pos->member.next, typeof(*pos), member);       \
104  -             prefetch(pos->member.next), &pos->member != (head);        \
105  +             &pos->member != (head);        \
106               pos = list_entry(pos->member.next, typeof(*pos), member))
107
108  /**
109 @@ -471,7 +467,7 @@ static inline void list_splice_tail_init(struct list_head *list,
110   */
111  #define list_for_each_entry_continue_reverse(pos, head, member)                \
112          for (pos = list_entry(pos->member.prev, typeof(*pos), member);       \
113  -             prefetch(pos->member.prev), &pos->member != (head);        \
114  +             &pos->member != (head);        \
115               pos = list_entry(pos->member.prev, typeof(*pos), member))
116
117  /**
118 @@ -483,7 +479,7 @@ static inline void list_splice_tail_init(struct list_head *list,
119   * Iterate over list of given type, continuing from current position.
120   */
121  #define list_for_each_entry_from(pos, head, member)                         \
122  -        for (; prefetch(pos->member.next), &pos->member != (head);        \
```

```
+        for (; &pos->member != (head);          \
                pos = list_entry(pos->member.next, typeof(*pos), member))

 /**
diff --git a/include/linux/rculist.h b/include/linux/rculist.h
index 900a97a..e3beb31 100644
--- a/include/linux/rculist.h
+++ b/include/linux/rculist.h
@@ -253,7 +253,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  */
 #define list_for_each_entry_rcu(pos, head, member) \
         for (pos = list_entry_rcu((head)->next, typeof(*pos), member); \
-                prefetch(pos->member.next), &pos->member != (head); \
+                &pos->member != (head); \
                 pos = list_entry_rcu(pos->member.next, typeof(*pos), member))


@@ -270,7 +270,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  */
 #define list_for_each_continue_rcu(pos, head) \
         for ((pos) = rcu_dereference_raw(list_next_rcu(pos)); \
-                prefetch((pos)->next), (pos) != (head); \
+                (pos) != (head); \
                 (pos) = rcu_dereference_raw(list_next_rcu(pos)))

 /**
@@ -284,7 +284,7 @@ static inline void list_splice_init_rcu(struct list_head *list,
  */
 #define list_for_each_entry_continue_rcu(pos, head, member)                 \
         for (pos = list_entry_rcu(pos->member.next, typeof(*pos), member); \
-             prefetch(pos->member.next), &pos->member != (head);        \
+             &pos->member != (head);        \
              pos = list_entry_rcu(pos->member.next, typeof(*pos), member))

 /**
--
2.1.3
```

**Listing 2:** The patch with which prefetching was removed from Linux implementation of linked lists.

# Appendix B
# Environment setup

During the experiments we used two machines with the setup described in table B.1.

**Table B.1:** Hardware and software specifications for lab-equipment.

| | **Machine 1** | **Machine 2** |
|---|---|---|
| CPU | Intel Core i7-3930K @ 3.2 GHz | Intel Core i7 @ 2.93 GHz |
| RAM | 64 GB | 16 GB |
| OS | Fedora 20 | Mageia 4 |
| Linux kernel | 3.16.6 | 3.14.18 |
| SSD | INTEL SSDSC2BW24 | N/A |
| HDD | WDC WD2003FZEX 7200 RPM | WDC WD5000AAKS 7200 RPM |

**Machine 1** was used as the database server and **Machine 2** was used to run the `pgbench` program issuing the queries over the network.
Throughout the project we used the following versions of the compilers:

- GCC 4.8.3
- Clang 3.4

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-01-16

**STUDENTER** Martin Lindblom, Fredrik Strandin
**HANDLEDARE** Jonas Skeppstedt (LTH)
**EXAMINATOR** Per Andersson (LTH)

# Prestandaanalys och optimering av PostgreSQL

POPULÄRVETENSKAPLIG SAMMANFATTNING **Martin Lindblom och Fredrik Strandin**

PostgreSQL är en populär databasserver som används i stora delar av industrin. Databasservrar används för att lagra, bearbeta och hämta uppgifter åt t.ex. företag, organisationer och universitet som de förlitar sig på i sitt arbete. Ofta blir dock dessa databaser flaskhalsen i hur snabbt arbete kan utföras, och därför har vi analyserat och förbättrat en populär databasserver.

Vad är det då som påverkar prestanda, och vad är enkelt att påverka? Vi har i vårt arbete valt att titta på två faktorer. Den första är filsystemet - hur informationen är lagrad på hårddisken. Det finns en uppsjö med filsystem som vänder sig till olika behov, vissa är väldigt enkla, andra har extremt komplicerade funktioner för att vara säker på att ens filer inte förstörs eller råkar försvinna. Det man ser här är att ju fler funktioner, desto långsammare blir filsystemen - i de flesta fallen. Dock fann vi ett undantag från denna regel i filsystemet *ext4*, som presterade bättre än sin föregångare *ext2*, trots nya funktioner som till exempel journalföring.

Den andra aspekten vi tittade på var kompilatorer och de optimeringar som de kan göra. En kompilator är det verktyg som översätter programmerarens kod till en binärfil - ettor och nollor som datorn kan förstå. I samband med detta kan kompilatorn göra optimeringar i programmet som gör att koden kör snabbare. Alla typer av optimeringar är dock inte bra för alla program, då olika program beter sig på olika sätt. Därför har man lagt optimeringarna i olika nivåer beroende på hur aggressivt de optimerar. Aggressiva optimeringar kan öka kodstorleken vilket kan försämra prestandan. Det finns även ett flertal olika kompilatorer som beter sig olika, där vi har valt att jämföra *GCC* och *Clang*. GCC är mer än 15 år äldre än Clang, vilket har sina för- och nackdelar. GCC är dåligt strukturerad vilket gör det svårt för utvecklarna att lägga till nya funktioner, men har å andra sidan använts väldigt länge och därmed blivit mer mogen. Det senare syntes tydligt i våra resultat som visade på att GCC fortfarande producerar bättre och snabbare program. Vår studie visar också att PostgreSQL är ett program som presterar bäst om man inte optimerar allt för aggressivt.

Vi ville också se om vi kunde få PostgreSQL att bli snabbare genom att ändra i koden. För att göra detta utnyttjar vi att moderna datorer har buffertar mellan arbetsminnet och processorn. Dessa buffertar kallas för cacheminnen och är extremt snabba att jobba med, men små eftersom dom är dyra att tillverka. Därför kan man hjälpa datorn att välja vad som ska hämtas in i cachen härnäst, så att den data man vill jobba med ligger redo att användas när den behövs. Dock är detta riskfyllt, och kräver att man har full koll på hur lång tid det tar att hämta datan. Gör man det för sent hjälper det inte, och ger konstiga effekter i cacheminnet som påverkar senare beräkningar negativt. Gör man det för tidigt så riskerar datan att skrivas över igen innan den ska användas. Efter analys av hur PostgreSQL beter sig, och hur det är skrivet, kunde vi slutligen hitta en del av koden där vi kunde ge denna typ av hjälp till datorn så att PostgreSQL blev 0.5% snabbare.

Sammanfattningsvis kan vi konstatera att PostgreSQL presterar bäst på ext4-filsystem, kompilerad med GCC på optimeringsnivå 2, samt att PostgreSQL kan fördelaktigt tipsa processorn om vad för data som ska användas. ■